Regular Expressions, *au point*

Andrea Asperti
Department of Computer Science,
University of Bologna
Mura Anteo Zamboni 7, 40127, Bologna,
ITALY
asperti@cs.unibo.it

Claudio Sacerdoti Coen
Department of Computer Science,
University of Bologna
Mura Anteo Zamboni 7, 40127, Bologna,
ITALY
sacerdot@cs.unibo.it

Enrico Tassi
Microsoft Research-INRIA Joint Center
enrico.tassi@inria.fr

Abstract

We introduce a new technique for constructing a finite state deterministic automaton from a regular expression, based on the idea of marking a suitable set of positions *inside* the expression, intuitively representing the possible points reached after the processing of an initial prefix of the input string. *Pointed* regular expressions join the elegance and the symbolic appealingness of Brzozowski's derivatives, with the effectiveness of McNaughton and Yamada's labelling technique, essentially combining the best of the two approaches.

Categories and Subject Descriptors F.1.1 [*Models of Computation*]

General Terms Theory

Keywords Regular expressions, Finite States Automata, Derivatives

1. Introduction

There is hardly a subject in Theoretical Computer Science that, in view of its relevance and elegance, has been so thoroughly investigated as the notion of *regular expression* and its relation with *finite state automata* (see e.g. [1, 2] for some recent surveys). All the studies in this area have been traditionally inspired by two precursory, basilar works: Brzozowski's theory of *derivatives* [3], and McNaughton and Yamada's algorithm [4]. The main advantages of derivatives are that they are syntactically appealing, easy to grasp and to prove correct (see [5] for a recent revisitation). On the other side, McNaughton and Yamada's approach results in a particularly efficient algorithm, still used by most pattern matchers like the popular grep and egrep utilities. The relation between the two approaches has been deeply investigated too, starting from the seminal work by Berry and Sethi [6] where it is shown how to refine Brzozowski's method to get to the efficient algorithm (Berry and Sethi' algorithm has been further improved by later authors [7, 8]).

Regular expressions are such small world that it is much at no one's surprise that all different approaches, at the end, turn out to be equivalent; still, their philosophy, their underlying intuition, and the techniques to be deployed can be sensibly different. Without having the pretension to say anything really original on the subject, we introduce in this paper a notion of *pointed* regular expression, that provides a cheap palliative for derivatives and allows a simple, direct and efficient construction of the deterministic finite automaton. Remarkably, the formal correspondence between pointed expressions and Brzozowski's derivatives is unexpectedly entangled (see Section 4.1) testifying the novelty and the not-so-trivial nature of the notion.

The idea of pointed expressions was suggested by an attempt of formalizing the theory of regular languages by means of an interactive prover[1]. At first, we started considering derivatives, since they looked more suitable to the kind of symbolic manipulations that can be easily dealt with by means of these tools. However, the need to consider *sets* of derivatives and, especially, to reason modulo associativity, commutativity and idempotence of sum, prompted us to look for an alternative notion. Now, it is clear that, in some sense, the derivative of a regular expression e is a set of "subexpressions" of e[2]: the only, crucial, difference is that we cannot forget their context. So, the natural solution is to *point* at subexpressions *inside* the original term. This immediately leads to the notion of *pointed* regular expression (pre), that is just a normal regular expression where some positions (it is enough to consider individual characters) have been pointed out. Intuitively, the points mark the positions inside the regular expression which have been reached after reading some prefix of the input string, or better the positions where the processing of the remaining string has to be started. Each pointed expression for e represents a state of the *deterministic* automaton associated with e; since we obviously have only a finite number of possible labellings, the number of states of the automaton is finite.

Pointed regular expressions allow the *direct* construction of the DFA [9] associated with a regular expression, in a way that is simple, intuitive, and efficient (the task is traditionally considered as *very involved* in the literature: see e.g [1], pag.71).

In the imposing bibliography on regular expressions - as far as we could discover - the only author mentioning a notion close to ours is Watson [10, 11]. However, he only deals with single points, while the most interesting properties of *pre* derive by their implicit additive nature (such as the possibility to compute the *move* operation by a single pass on the marked expression: see definition 21).

[1] The rule of the game was to avoid overkilling, i.e. not make it more complex than deserved.

[2] This is also the reason why, at the end, we only have a finite number of derivatives.

2. Regular expressions

DEFINITION 1. *A regular expression over the alphabet Σ is an expression e generated by the following grammar:*

$$E ::= \emptyset|\epsilon|a|E+E|EE|E^*$$

with $a \in \Sigma$

DEFINITION 2. *The language $L(e)$ associated with the regular expression e is defined by the following rules:*

$$\begin{array}{rcl} L(\emptyset) & = & \emptyset \\ L(\epsilon) & = & \{\epsilon\} \\ L(a) & = & \{a\} \\ L(e_1+e_2) & = & L(e_1) \cup L(e_2) \\ L(e_1 e_2) & = & L(e_1) \cdot L(e_2) \\ L(e^*) & = & L(e)^* \end{array}$$

where ϵ is the empty string, $L_1 \cdot L_2 = \{\, l_1 l_2 \mid l_1 \in L_1,\ l_2 \in L_2\}$ is the concatenation of L_1 and L_2 and L^* is the so called Kleene's closure of L: $L^* = \bigcup_{i=0}^{\infty} L^i$, with $L^0 = \epsilon$ and $L^{i+1} = L \cdot L^i$.

DEFINITION 3 (nullable).
A regular expression e is said to be nullable if $\epsilon \in L(e)$.

The fact of being nullable is decidable; it is easy to prove that the characteristic function $\nu(e)$ can be computed by the following rules:

$$\begin{array}{rcl} \nu(\emptyset) & = & \mathit{false} \\ \nu(\epsilon) & = & \mathit{true} \\ \nu(a) & = & \mathit{false} \\ \nu(e_1+e_2) & = & \nu(e_1) \vee \nu(e_2) \\ \nu(e_1 e_2) & = & \nu(e_1) \wedge \nu(e_2) \\ \nu(e^*) & = & \mathit{true} \end{array}$$

DEFINITION 4. *A deterministic finite automaton (DFA) is a quintuple (Q, Σ, q_0, t, F) where*

- *Q is a finite set of states;*
- *Σ is the input alphabet;*
- *$q_0 \in Q$ is the initial state;*
- *$t : Q \times \Sigma \to Q$ is the state transition function;*
- *$F \subseteq Q$ is the set of final states.*

The transition function t is extended to strings in the following way:

DEFINITION 5. *Given a function $t : Q \times \Sigma \to Q$, the function $t^* : Q \times \Sigma^* \to Q$ is defined as follows:*

$$t^*(q,w) = \begin{cases} t(q,\epsilon) = q \\ t(q,aw') = t^*(t(q,a),w') \end{cases}$$

DEFINITION 6. *Let $A = (Q, \Sigma, q_0, t, F)$ be a DFA; the language recognized A is defined as follows:*

$$L(A) = \{w | t^*(q_0,w) \in F\}$$

3. Pointed regular expressions

DEFINITION 7.

1. *A* pointed item *over the alphabet Σ is an expression e generated by following grammar:*

$$E ::= \emptyset|\epsilon|a| \bullet a|E+E|EE|E^*$$

   with $a \in \Sigma$;
2. *A* pointed regular expression *(pre) is a pair $\langle e, b\rangle$ where b is a boolean and e is a pointed item.*

The term $\bullet a$ is used to point to a position inside the regular expression, preceding the given occurrence of a. In a pointed regular expression, the boolean must be intuitively understood as the possibility to have a trailing point at the end of the expression.

DEFINITION 8. *The* carrier *$|e|$ of an item e is the regular expression obtained from e by removing all the points. Similarly, the* carrier *of a pointed regular expression is the carrier of its item.*

In the sequel, we shall often use the same notation for functions defined over items or pres, leaving to the reader the simple disambiguation task. Moreover, we use the notation $\epsilon(b)$, where b is a boolean, with the following meaning:

$$\epsilon(\mathit{true}) = \{\epsilon\} \qquad \epsilon(\mathit{false}) = \emptyset$$

DEFINITION 9.

1. *The language $L_p(e)$ associated with the item e is defined by the following rules:*

$$\begin{array}{rcl} L_p(\emptyset) & = & \emptyset \\ L_p(\epsilon) & = & \emptyset \\ L_p(a) & = & \emptyset \\ L_p(\bullet a) & = & \{a\} \\ L_p(e_1+e_2) & = & L_p(e_1) \cup L_p(e_2) \\ L_p(e_1 e_2) & = & L_p(e_1) \cdot L(|e_2|) \cup L_p(e_2) \\ L_p(e^*) & = & L_p(e) \cdot L(|e|^*) \end{array}$$

2. *For a pointed regular expression $\langle e, b\rangle$ we define*

$$L_p(\langle e, b\rangle) = L_p(e) \cup \epsilon(b)$$

EXAMPLE 10.

$$L_p((a + \bullet b)^*) = L(b(a+b)^*)$$

Indeed,

$$\begin{array}{l} L_p((a+\bullet b)^*) = \\ \quad = L_p(a+\bullet b) \cdot L(|a+\bullet b|^*) \\ \quad = (L_p(a) \cup L_p(\bullet b)) \cdot L((a+b)^*) \\ \quad = \{b\} \cdot L((a+b)^*) \\ \quad = L(b(a+b)^*) \end{array}$$

Let us incidentally observe that, as shown by the previous example, pointed regular expressions can provide a more compact syntax for denoting languages than traditional regular expressions. This may have important applications to the investigation of the descriptional complexity (succinctness) of regular languages (see e.g. [12, 13, 14]).

EXAMPLE 11. *If e contains no point (i.e. $e = |e|$) then $L_p(e) = \emptyset$*

LEMMA 12. *If e is a pointed item then $\epsilon \notin L_p(e)$. Hence, $\epsilon \in L_p(\langle e, b\rangle)$ if and only if $b = \mathit{true}$.*

Proof. A trivial structural induction on e.

3.1 Broadcasting points

Intuitively, a regular expression e must be understood as a pointed expression with a single point in front of it. Since however we only allow points over symbols, we must broadcast this initial point inside the expression, that essentially corresponds to the ϵ-closure operation on automata. We use the notation $\bullet(\cdot)$ to denote such an operation.

The broadcasting operator is also required to lift the item constructors (choice, concatenation and Kleene's star) from items to pres: for example, to concatenate a pre $\langle e_1, \mathit{true}\rangle$ with another pre $\langle e_2, b_2\rangle$, we must first broadcast the trailing point of the first expression inside e_2 and then pre-pend e_1; similarly for the star operation. We could define first the broadcasting function $\bullet(\cdot)$ and then the lifted constructors; however, both the definition and the theory of the broadcasting function are simplified by making it co-recursive with the lifted constructors.

DEFINITION 13.

1. *The function $\bullet(\cdot)$ from pointed item to pres is defined as follows:*

$$\begin{array}{rcl}
\bullet(\emptyset) & = & \langle \emptyset, \mathit{false}\rangle \\
\bullet(\epsilon) & = & \langle \epsilon, \mathit{true}\rangle \\
\bullet(a) & = & \langle \bullet a, \mathit{false}\rangle \\
\bullet(\bullet a) & = & \langle \bullet a, \mathit{false}\rangle \\
\bullet(e_1+e_2) & = & \bullet(e_1) \oplus \bullet(e_2) \\
\bullet(e_1 e_2) & = & \bullet(e_1) \odot \langle e_2, \mathit{false}\rangle \\
\bullet(e^*) & = & \langle e'^*, \mathit{true}\rangle \text{ where } \bullet(e) = \langle e', b'\rangle
\end{array}$$

2. *The lifted constructors are defined as follows*

$$\begin{array}{rl}
\langle e_1', b_1'\rangle \oplus \langle e_2', b_2'\rangle = & \langle e_1+e_2, b_1' \vee b_2'\rangle \\
\langle e_1', b_1'\rangle \odot \langle e_2', b_2'\rangle = & \begin{cases} \langle e_1' e_2', b_2'\rangle & \text{when } b_1' = \mathit{false} \\ \langle e_1' e_2'', b_2' \vee b_2''\rangle & \text{when } b_1' = \mathit{true} \\ & \text{and } \bullet(e_2') = \langle e_2'', b_2''\rangle \end{cases} \\
\langle e', b'\rangle^\star = & \begin{cases} \langle e'^*, \mathit{false}\rangle & \text{when } b' = \mathit{false} \\ \langle e''^*, \mathit{true}\rangle & \text{when } b' = \mathit{true} \\ & \text{and } \bullet(e') = \langle e'', b''\rangle \end{cases}
\end{array}$$

The apparent complexity of the previous definition should not hide the extreme simplicity of the broadcasting operation: on a sum we proceed in parallel; on a concatenation e_1e_2, we first work on e_1 and in case we reach its end we pursue broadcasting inside e_2; in case of e^* we broadcast the point inside e recalling that we shall eventually have a trailing point.

EXAMPLE 14. *Suppose to broadcast a point inside*

$$(a+\epsilon)(b^*a+b)b$$

We start working in parallel on the first occurrence of a (where the point stops), and on ϵ that gets traversed. We have hence reached the end of $a+\epsilon$ and we must pursue broadcasting inside $(b^*a+b)b$. Again, we work in parallel on the two additive subterms b^*a and b; the first point is allowed to both enter the star, and to traverse it, stopping in front of a; the second point just stops in front of b. No point reached that end of b^*a+b hence no further propagation is possible. In conclusion:

$$\bullet((a+\epsilon)(b^*a+b)b) = (\bullet a+\epsilon)((\bullet b)^* \bullet a + \bullet b)b$$

DEFINITION 15. *The broadcasting function is extended to pres in the obvious way:*

$$\bullet(\langle e, b\rangle) = \langle e', b \vee b'\rangle \text{ where } \bullet(e) = \langle e', b'\rangle$$

As we shall prove in Corollary 18, broadcasting an initial point may reach the end of an expression e if and only if e is nullable.
The following theorem characterizes the broadcasting function and also shows that the semantics of the lifted constructors on pres is coherent with the corresponding constructors on items.

THEOREM 16.

1. $L_p(\bullet e) = L_p(e) \cup L(|e|)$.
2. $L_p(e_1 \oplus e_2) = L_p(e_1) \cup L_p(e_2)$
3. $L_p(e_1 \odot e_2) = L_p(e_1) \cdot L(|e_2|) \cup L_p(e_2)$
4. $L_p(e^\star) = L_p(e) \cdot L(|e|)^*$

We do first the proof of 2., followed by the simultaneous proof of 1. and 3., and we conclude with the proof of 4.

Proof.[of 2.] We need to prove $L_p(e_1 \oplus e_2) = L_p(e_1) \cup L_p(e_2)$.

$$\begin{array}{l}
L_p(\langle e_1', b_1'\rangle \oplus \langle e_2', b_2'\rangle) = \\
\quad = L_p(\langle e_1'+e_2', b_1' \vee b_2'\rangle) \\
\quad = L_p(e_1'+e_2') \cup \epsilon(b_1') \cup \epsilon(b_2') \\
\quad = L_p(e_1') \cup \epsilon(b_1') \cup L_p(e_2') \cup \epsilon(b_2') \\
\quad = L_p(e_1) \cup L_p(e_2)
\end{array}$$

Proof.[of 1. and 3.] We prove 1. ($L_p(\bullet e) = L_p(e) \cup L(|e|)$) by induction on the structure of e, assuming that 3. holds on terms structurally smaller than e.

- $L_p(\bullet(\emptyset)) = L_p(\langle\emptyset, \mathit{false}\rangle) = \emptyset = L_p(\emptyset) \cup L(|\emptyset|)$.
- $L_p(\bullet(\epsilon)) = L_p(\langle\epsilon, \mathit{true}\rangle) = \{\epsilon\} = L_p(\epsilon) \cup L_p(|\epsilon|)$.
- $L_p(\bullet(a)) = L_p(\langle a, \mathit{false}\rangle) = \{a\} = L_p(a) \cup L(|a|)$.
- $L_p(\bullet(\bullet a)) = L_p(\langle \bullet a, \mathit{false}\rangle) = \{a\} = L_p(\bullet a) \cup L(|\bullet a|)$.
- Let $e = e_1 + e_2$. By induction hypothesis we know that

  $$L_p(\bullet(e_i)) = L_p(e_i) \cup L(|e_i|)$$

  Thus, by 2., we have

  $$\begin{array}{l}
  L_p(\bullet(e_1+e_2)) = \\
  \quad = L_p(\bullet(e_1) \oplus \bullet(e_2)) \\
  \quad = L_p(\bullet(e_1)) \cup L_p(\bullet(e_2)) \\
  \quad = L_p(e_1) \cup L(|e_1|) \cup L_p(e_2) \cup L(|e_2|) \\
  \quad = L_p(e_1+e_2) \cup L(|e_1+e_2|)
  \end{array}$$

- Let $e = e_1e_2$. By induction hypothesis we know that

  $$L_p(\bullet(e_i)) = L_p(e_i) \cup L(|e_i|)$$

  Thus, by 3. over the structurally smaller terms e_1 and e_2

  $$\begin{array}{l}
  L_p(\bullet(e_1e_2)) = \\
  \quad = L_p(\bullet(e_1) \odot \langle e_2, \mathit{false}\rangle) \\
  \quad = L_p(\bullet(e_1)) \cdot L(|e_2|) \cup L_p(e_2) \\
  \quad = (L_p(e_1) \cup L(|e_1|)) \cdot L(|e_2|) \cup L_p(e_2) \\
  \quad = L_p(e_1) \cdot L(|e_2|) \cup L(|e_1|) \cdot L(|e_2|) \cup L_p(e_2) \\
  \quad = L_p(e_1e_2) \cup L(|e_1e_2|)
  \end{array}$$

- Let $e = e_1^*$. By induction hypothesis we know that

  $$L_p(\bullet(e_1)) = L_p(e_1') \cup \epsilon(b_1') = L_p(e_1) \cup L(|e_1|)$$

  and in particular, since by Lemma 12 $\epsilon \notin L_p(e_1)$,

  $$L_p(e_1') = L_p(e_1) \cup (L(|e_1|) \setminus \epsilon(b_1'))$$

  Then,

  $$\begin{array}{l}
  L_p(\bullet(e_1^*)) = \\
  \quad = L_p(\langle e_1'^*, \mathit{true}\rangle) \\
  \quad = L_p(e_1'^*) \cup \epsilon \\
  \quad = L_p(e_1')L(|e_1^*|) \cup \epsilon \\
  \quad = (L_p(e_1) \cup (L(|e_1|) \setminus \epsilon(b_1')))L(|e_1^*|) \cup \epsilon \\
  \quad = L_p(e_1)L(|e_1^*|) \cup (L(|e_1|) \setminus \epsilon(b_1'))L(|e_1^*|) \cup \epsilon \\
  \quad = L_p(e_1)L(|e_1^*|) \cup L(|e_1^*|) \\
  \quad = L_p(e_1^*) \cup L(|e_1^*|)
  \end{array}$$

Having proved 1. for e assuming that 3. holds on terms structurally smaller than e, we now assume that 1. holds for e_1 and e_2 in order to prove 3.: $L_p(e_1 \odot e_2) = L_p(e_1) \cdot L(|e_2|) \cup L_p(e_2)$
We distinguish the two cases of the definition of $\odot$:

$$
\begin{aligned}
&L_p(\langle e'_1, \mathit{false}\rangle \odot \langle e'_2, b'_2\rangle) = \\
&\quad = L_p(\langle e'_1 e'_2, b'_2\rangle) \\
&\quad = L_p(e'_1 e'_2) \cup \epsilon(b'_2) \\
&\quad = L_p(e'_1) \cdot L(|e'_2|) \cup L_p(e'_2) \cup \epsilon(b'_2) \\
&\quad = L_p(e_1) \cdot L(|e_2|) \cup L_p(e_2)
\end{aligned}
$$

$$
\begin{aligned}
&L_p(\langle e'_1, \mathit{true}\rangle \odot \langle e'_2, b'_2\rangle) = \\
&\quad = L_p(\langle e'_1 e''_2, b'_2 \vee b''_2\rangle) \\
&\quad = L_p(e'_1 e''_2) \cup \epsilon(b'_2) \cup \epsilon(b''_2) \\
&\quad = L_p(e'_1) \cdot L(|e''_2|) \cup L_p(e''_2) \cup \epsilon(b'_2) \cup \epsilon(b''_2) \\
&\quad = L_p(e'_1) \cdot L(|e''_2|) \cup L_p(e'_2) \cup L(|e'_2|) \cup \epsilon(b'_2) \\
&\quad = (L_p(e'_1) \cup \epsilon(\mathit{true})) \cdot L(|e_2|) \cup L_p(e'_2) \cup \epsilon(b'_2) \\
&\quad = L_p(e_1) \cdot L(|e_2|) \cup L_p(e_2)
\end{aligned}
$$

Proof.[of 4.] We need to prove $L_p(e^\star) = L_p(e) \cdot L(|e|)^*$. We distinguish the two cases of the definition of $\cdot^\star$:

$$
\begin{aligned}
&L_p(\langle e', \mathit{false}\rangle^\star) = \\
&\quad = L_p(\langle e'^*, \mathit{false}\rangle) \\
&\quad = L_p(e'^*) \\
&\quad = L_p(e') \cdot L(|e'|)^* \\
&\quad = (L_p(e') \cup \epsilon(\mathit{false})) \cdot L(|e'|)^* \\
&\quad = L_p(e) \cdot L(|e|)^*
\end{aligned}
$$

$$
\begin{aligned}
&L_p(\langle e', \mathit{true}\rangle^\star) = \\
&\quad = L_p(\langle e''^*, \mathit{true}\rangle) \cup \epsilon \\
&\quad = L_p(e''^*) \cup \epsilon \\
&\quad = L_p(e'') \cdot L(|e''|)^* \cup \epsilon \\
&\quad = (L_p(e') \cup L(|e'|)) \cdot L(|e''|)^* \cup \epsilon \\
&\quad = L_p(e') \cdot L(|e''|) \cup L(|e'|) \cdot L(|e''|)^* \cup \epsilon \\
&\quad = L_p(e') \cdot L(|e''|) \cup L(|e'|)^* \\
&\quad = (L_p(e') \cup \epsilon(\mathit{true})) \cdot L(|e''|) \\
&\quad = L_p(e) \cdot L(|e|)^*
\end{aligned}
$$

COROLLARY 17. *For any regular expression e,* $L(e) = L_p(\bullet e)$.

Another important corollary is that an initial point reaches the end of a (pointed) expression e if and only if e is able to generate the empty string.

COROLLARY 18. $\bullet e = \langle e', \mathit{true}\rangle$ *if and only if* $\epsilon \in L(|e|)$.

Proof. By theorem 16 we know that $L_p(\bullet e) = L_p(e) \cup L(|e|)$. So, if $\epsilon \in L_p(\bullet e)$, since by Lemma 12 $\epsilon \notin L_p(e)$, it must be $\epsilon \in L(|e|)$. Conversely, if $\epsilon \in L(|e|)$ then $\epsilon \in L_p(\bullet e)$; if $\bullet e = \langle e', b\rangle$, this is possible only provided $b = \mathit{true}$.

To conclude this section, let us prove the idempotence of the $\bullet(\cdot)$ function (it will only be used in Section 5, and can be skipped at a first reading). To this aim we need a technical lemma whose straightforward proof by case analysis is omitted.

LEMMA 19. 1. $\bullet(e_1 \oplus e_2) = \bullet(e_1) \oplus \bullet(e_2)$
2. $\bullet(e_1 \odot e_2) = \bullet(e_1) \odot e_2$

THEOREM 20. $\bullet(\bullet(e)) = \bullet(e)$

Proof. The proof is by induction on e.

- $\bullet(\bullet(\emptyset)) = \bullet(\langle \emptyset, \mathit{false}\rangle) = \langle \emptyset, \mathit{false} \vee \mathit{false}\rangle = \bullet(\emptyset)$
- $\bullet(\bullet(\epsilon)) = \bullet(\langle \epsilon, \mathit{true}\rangle) = \langle \epsilon, \mathit{true} \vee \mathit{true}\rangle = \bullet(\epsilon)$
- $\bullet(\bullet(a)) = \bullet(\langle \bullet a, \mathit{false}\rangle) = \langle \bullet a, \mathit{false} \vee \mathit{false}\rangle = \bullet(a)$
- $\bullet(\bullet(\bullet a)) = \bullet(\langle \bullet a, \mathit{false}\rangle) = \langle \bullet a, \mathit{false} \vee \mathit{false}\rangle = \bullet(\bullet a)$
- If e is $e_1 + e_2$ then
$$
\begin{aligned}
&\bullet(\bullet(e_1 + e_2)) = \bullet(\bullet(e_1) \oplus \bullet(e_2)) = \bullet(\bullet(e_1)) \oplus \bullet(\bullet(e_2)) = \\
&\quad = \bullet(e_1) \oplus \bullet(e_2) = \bullet(e_1 + e_2)
\end{aligned}
$$
- If e is $e_1 e_2$ then
$$
\begin{aligned}
&\bullet(\bullet(e_1 e_2)) = \bullet(\bullet(e_1) \odot \langle e_2, \mathit{false}\rangle) \bullet (\bullet(e_1)) \odot \langle e_2, \mathit{false}\rangle = \\
&\quad = \bullet(e_1) \odot \langle e_2, \mathit{false}\rangle = \bullet(e_1 e_2)
\end{aligned}
$$
- If e is e_1^*, let $\bullet(e_1) = \langle e', b'\rangle$ and let $\bullet(e') = \langle e'', b''\rangle$. By induction hypothesis,
$$\langle e', b'\rangle = \bullet(e_1) = \bullet(\bullet(e_1)) = \bullet(\langle e', b'\rangle) = \langle e'', b' \vee b''\rangle$$
and thus $e' = e''$. Finally
$$
\begin{aligned}
&\bullet(\bullet(e_1^*)) = \bullet(\langle e'^*, \mathit{true}\rangle) = \langle e''^*, \mathit{true} \vee b''\rangle = \langle e'^*, \mathit{true}\rangle = \\
&\quad = \bullet(e_1^*)
\end{aligned}
$$

3.2 The move operation

We now define the move operation, that corresponds to the advancement of the state in response to the processing of an input character a. The intuition is clear: we have to look at points inside e preceding the given character a, let the point traverse the character, and broadcast it. All other points must be removed.

DEFINITION 21.

1. *The function* $\mathit{move}(e, a)$ *taking in input a pointed item* e*, a character* $a \in \Sigma$ *and giving back a pointer regular expression is defined as follow, by induction on the structure of* e*:*
$$
\begin{array}{rcl}
\mathit{move}(\emptyset, a) &=& \langle \emptyset, \mathit{false}\rangle \\
\mathit{move}(\epsilon, a) &=& \langle \epsilon, \mathit{false}\rangle \\
\mathit{move}(b, a) &=& \langle b, \mathit{false}\rangle \\
\mathit{move}(\bullet a, a) &=& \langle a, \mathit{true}\rangle \\
\mathit{move}(\bullet b, a) &=& \langle b, \mathit{false}\rangle \ \mathit{if}\ b \neq a \\
\mathit{move}(e_1 + e_2, a) &=& \mathit{move}(e_1, a) \oplus \mathit{move}(e_2, a) \\
\mathit{move}(e_1 e_2, a) &=& \mathit{move}(e_1, a) \odot \mathit{move}(e_2, a) \\
\mathit{move}(e^*, a) &=& \mathit{move}(e, a)^\star
\end{array}
$$
2. *The move function is extended to pres by just ignoring the trailing point:* $\mathit{move}(\langle e, b\rangle, a) = \mathit{move}(e, a)$

EXAMPLE 22. *Let us consider the pre* $(\bullet a + \epsilon)((\bullet b)^* \bullet a + \bullet b)b$ *and the two moves w.r.t. the characters* a *and* b*. For* a*, we have two possible positions (all other points gets erased); the innermost point stops in front of the final* b*, the other one broadcast inside* $(b^* a + b)b$*, so*

$$\mathit{move}((\bullet a + \epsilon)((\bullet b)^* \bullet a + \bullet b)b, a) = \langle (a + \epsilon)((\bullet b)^* \bullet a + \bullet b) \bullet b, \mathit{false}\rangle$$

For b*, we have two positions too. The innermost point still stops in front of the final* b*, while the other point reaches the end of* b^* *and must go back through* $b^* a$*:*

$$\mathit{move}((\bullet a + \epsilon)((\bullet b)^* \bullet a + \bullet b) \bullet b, b) = \langle (a + \epsilon)((\bullet b)^* \bullet a + b) \bullet b, \mathit{false}\rangle$$

THEOREM 23. *For any pointed regular expression* e *and string* w*,*

$$w \in L_p(\mathit{move}(e, a)) \Leftrightarrow aw \in L_p(e)$$

Proof. The proof is by induction on the structure of e.

- if e is atomic, and e is not a pointed symbol, then both $L_p(\mathit{move}(e, a))$ and $L_p(e)$ are empty, and hence both sides are false for any w;
- if $e = \bullet a$ then $L_p(\mathit{move}(\bullet a, a)) = L_p(\langle a, \mathit{true}\rangle) = \{\epsilon\}$ and $L_p(\bullet a) = \{a\}$;
- if $e = \bullet b$ with $b \neq a$ then $L_p(\mathit{move}(\bullet b, a)) = L_p(\langle b, \mathit{false}\rangle) = \emptyset$ and $L_p(\bullet b) = \{b\}$; hence for any string w, both sides are false;
- if $e = e_1 + e_2$ by induction hypothesis $w \in L_p(\mathit{move}(e_i, a)) \Leftrightarrow aw \in L_p(e_i)$, hence,

$$\begin{aligned}
&w \in L_p(move(e_1+e_2,a)) \Leftrightarrow\\
&\quad\Leftrightarrow w \in L_p(move(e_1,a) \oplus move(e_2,a))\\
&\quad\Leftrightarrow w \in L_p(move(e_1,a)) \cup L_p(move(e_2,a))\\
&\quad\Leftrightarrow (w \in L_p(move(e_1,a))) \vee (w \in L_p(move(e_2,a)))\\
&\quad\Leftrightarrow (aw \in L_p(e_1)) \vee (aw \in L_p(e_2))\\
&\quad\Leftrightarrow aw \in L_p(e_1) \cup L_p(e_2)\\
&\quad\Leftrightarrow aw \in L_p(e_1+e_2)
\end{aligned}$$

– suppose $e = e_1e_2$, by induction hypothesis $w \in L_p(move(e_i,a)) \Leftrightarrow aw \in L_p(e_i)$, hence,

$$\begin{aligned}
&w \in L_p(move(e_1e_2,a)) \Leftrightarrow\\
&\quad\Leftrightarrow w \in L_p(move(e_1,a) \odot move(e_2,a))\\
&\quad\Leftrightarrow w \in L_p(move(e_1,a)) \cdot L|e_2| \cup L_p(move(e_2,a))\\
&\quad\Leftrightarrow w \in L_p(move(e_1,a)) \cdot L|e_2| \vee w \in L_p(move(e_2,a))\\
&\quad\Leftrightarrow (\exists w_1,w_2, w = w_1w_2 \wedge w_1 \in L_p(move(e_1,a))\\
&\qquad\wedge w_2 \in L(|e_2|)) \vee w \in L_p(move(e_2,a))\\
&\quad\Leftrightarrow (\exists w_1,w_2, w = w_1w_2 \wedge aw_1 \in L_p(e)\\
&\qquad\wedge w_2 \in L(|e_2|)) \vee aw \in L_p(e_2)\\
&\quad\Leftrightarrow (aw \in L_p(e_1) \cdot L|e_2|) \vee (aw \in L_p(e_2))\\
&\quad\Leftrightarrow aw \in L_p(e_1) \cdot L|e_2| \cup \in L_p(e_2)\\
&\quad\Leftrightarrow aw \in L_p(e_1e_2)
\end{aligned}$$

– suppose $e = e_1^*$, by induction hypothesis $w \in L_p(move(e_1,a)) \Leftrightarrow aw \in L_p(e_1)$, hence,

$$\begin{aligned}
&w \in L_p(move(e_1^*,a)) \Leftrightarrow\\
&\quad\Leftrightarrow w \in L_p(move(e_1,a))^\star\\
&\quad\Leftrightarrow w \in L_p(move(e_1,a)) \cdot L(|move(e_1,a)|)^*\\
&\quad\Leftrightarrow \exists w_1,w_2, w = w_1w_2 \wedge w_1 \in L_p(move(e_1,a))\\
&\qquad\wedge w_2 \in L(|e_1|)^*\\
&\quad\Leftrightarrow \exists w_1,w_2, w = w_1w_2 \wedge aw_1 \in L_p(e_1) \wedge w_2 \in L(|e_1|)^*\\
&\quad\Leftrightarrow aw \in L_p(e_1) \cdot L(|e_1|)^*\\
&\quad\Leftrightarrow aw \in L_p(e_1^*)
\end{aligned}$$

We extend the move operations to strings as usual.

DEFINITION 24.

$$move^*(e,\epsilon) = e \qquad move^*(e,aw) = move^*(move(e,a),w)$$

THEOREM 25. *For any pointed regular expression e and all strings α, β,*

$$\beta \in L_p(move^*(e,\alpha)) \Leftrightarrow \alpha\beta \in L_p(e)$$

Proof. A trivial induction on the length of α, using theorem 23.

COROLLARY 26. *For any pointed regular expression e and any string α,*

$$\alpha \in L_p(e) \Leftrightarrow \exists e', L_p(move^*(e,\alpha)) = \langle e', true\rangle$$

Proof. By Theorems 25 and Lemma 12.

3.3 From regular expressions to DFAs

DEFINITION 27. *To any regular expression e we may associate a DFA $D_e = (Q, \Sigma, q_0, t, F)$ defined in the following way:*

- *Q is the set of all possible pointed expressions having e as carrier;*
- *Σ is the alphabet of the regular expression*
- *q_0 is $\bullet e$;*
- *t is the move operation of definition 21;*
- *F is the subset of pointed expressions $\langle e, b\rangle$ with $b = true$.*

THEOREM 28. $\quad L(D_e) = L(e)$

Proof. By definition,

$$w \in L(D_e) \leftrightarrow move^*(\bullet(e), w) = \langle e', true\rangle$$

for some e'. By the previous theorem, this is possible if an only if $w \in L_p(\bullet(e))$, and by corollary 17, $L_p(\bullet(e)) = L(e)$.

REMARK 29. *The fact that the set Q of states of D_e is finite is obvious: its cardinality is at most 2^{n+1} where n is the number of symbols in e. This is one of the advantages of pointed regular expressions w.r.t. derivatives, whose finite nature only holds after a suitable quotient, and is a relatively complex property to prove (see [3]).*

The automaton D_e just defined may have many inaccessible states. We can provide another algorithmic and direct construction that yields the same automaton restricted to the accessible states only.

DEFINITION 30. *Let e be a regular expression and let q_0 be $\bullet e$. Let also*

$$\begin{aligned}
Q_0 &:= \{q_0\}\\
Q_{n+1} &:= Q_n \cup \{e' | e' \notin Q_n \wedge \exists a. \exists e \in Q_n. move(e,a) = e'\}
\end{aligned}$$

Since every Q_n is a subset of the finite set of pointed regular expressions, there is an m such that $Q_{m+1} = Q_m$. We associate to e the DFA $D_e = (Q_m, \Sigma, q_0, F, t)$ where F and t are defined as for the previous construction.

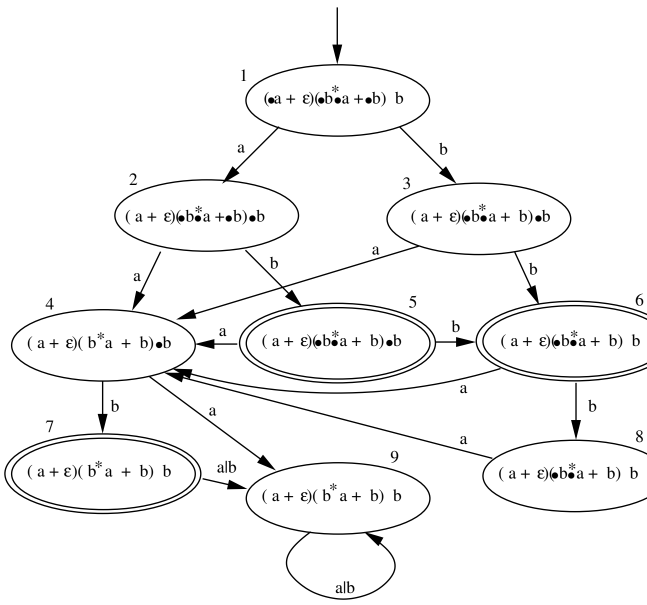

Figure 1. DFA for $(a+\epsilon)(b^*a+b)b$

In Figure 1 we describe the DFA associated with the regular expression $(a+\epsilon)(b^*a+b)b$. The graphical description of the automaton is the traditional one, with nodes for states and labelled arcs for transitions. Unreachable states are not shown. Final states are emphasized by a double circle: since a state $\langle e, b\rangle$ is final if and only if b is true, we may just label nodes with the item (for instance, the pair of states $6-8$ and $7-9$ only differ for the fact that 6 and 7 are final, while 8 and 9 are not).

3.4 Admissible relations and minimization

The automaton in Figure 1 is minimal. This is not always the case. For instance, for the expression $(ac+bc)^*$ we obtain the automaton of Figure 2, and it is easy to see that the two states corresponding to the pres $(a \bullet c + bc)^*$ and $(ac + b \bullet c)^*$ are equivalent (a way to prove it is to observe that they define the same language).
The latter remark, motivates the following definition.

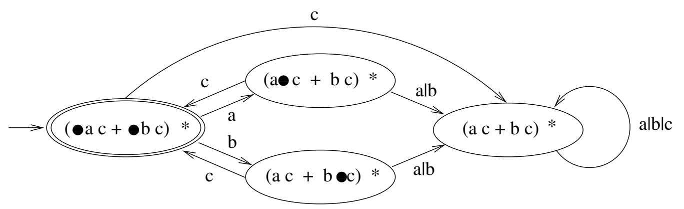

Figure 2. DFA for $(ac + bc)^*$

DEFINITION 31. *An equivalence relation $\approx$ over pres having the same carrier is admissible when for all e_1 and e_2*

- *if $e_1 \approx e_2$ then $L_p(e_1) = L_p(e_2)$*
- *if $e_1 \approx e_2$ then for all a $move(e_1, a) \approx move(e_2, a)$*

DEFINITION 32. *To any regular expression e and admissible equivalence relation over pres over e, we can directly associate the DFA $D_e/{\approx} = (Q/{\approx}, \Sigma, [q_0]_{\approx}, move^*/{\approx}, F/{\approx})$ where $move^*/{\approx}$ is the $move^*$ operation lifted to equivalence classes thanks to the second admissibility condition.*

In place of working with equivalence classes, for formalization and implementation purposes it is simpler to work on representative of equivalence classes. Instead of choosing a priori a representative of each equivalence class, we can slightly modify the algorithmic construction of definition 30 so that it dynamically identifies the representative of the equivalence classes. It is sufficient to read each element of Q_n as a representative of its equivalence class and to change the test $e' \notin Q_n$ so that the new state e' is compared to the representatives in Q_n up to $\approx$:

DEFINITION 33. *In definition 30 change the definition of Q_{n+1} as follows:*

$$Q_{n+1} := \\ Q_n \cup \{e' | \exists a.\exists e \in Q_n.move(e, a) = e' \wedge \nexists e'' \in Q_n.e' \approx e''\}$$

The transition function t is defined as $t(e, a) = e'$ where $move(e, a) = e''$ and e' is the unique state of Q_m such that $e' \equiv e''$.

In an actual implementation, the transition function t is computed together with the sets Q_n at no additional cost.

THEOREM 34. *Replacing each state e of the automaton of definition 33 with $[e]/{\approx}$, we obtain the restriction of the automaton of definition 32 to the accessible states.*

We still need to prove that quotienting over $\approx$ does not change the language recognized by the automaton.

THEOREM 35. $L(D_e/{\approx}) = L(e)$

Proof. By theorem 28, it is sufficient to prove $L(D_e) = L(D_e/{\approx})$ or, equivalently, that for all w, $move^*/{\approx}([q_0]/{\approx}, w) \in F/{\approx} \iff move^*(q_0, w) \in F$. We show this to hold by proving by induction over w that for all q

$$[move^*(q, w)]/{\approx} = move^*/{\approx}([q]/{\approx}, w)$$

Base case: $move^*/{\approx}([q]/{\approx}, \epsilon) = [q]/{\approx} = [move^*(q, \epsilon)]/{\approx}$
Inductive step: by condition (2) of admissibility, for all $q_1 \in [q_0]/{\approx}$, we have $move(q_1, a) \approx move(q_0, a)$ and thus

$$move/{\approx}([q_0]/{\approx}, a) = [move(q_0, a)]/{\approx}$$

$$\begin{aligned}\text{Hence}\quad & move^*/{\approx}([q_0]/{\approx}, aw) = \\ & = move^*/{\approx}(move/{\approx}([q_0]/{\approx}, a), w) \\ & = move^*/{\approx}([move(q_0, a)]/{\approx}, w) \\ & = [move^*(move(q_0, a), w)]/{\approx} \\ & = [move^*(q_0, aw)]/{\approx}\end{aligned}$$

The set of admissible equivalence relations over e is a bounded lattice, ordered by refinement, whose bottom element is syntactic identity and whose top element is $e_1 \approx e_2$ iff $L(e_1) = L(e_2)$. Moreover, if $\approx_1 < \approx_2$ (the first relation is a strict refinement of the second one), the number of states of $D_e/{\approx_1}$ is strictly larger than the number of states of $D_e/{\approx_2}$.

THEOREM 36. *If $\approx$ is the top element of the lattice, than $D_e/{\approx}$ is the minimal automaton that recognizes $L(e)$.*

Proof. By the previous theorem, $D_e/{\approx}$ recognizes $L(e)$ and has no unreachable states. By absurd, let $D' = (Q', \Sigma', q_0', t', F')$ be another smaller automaton that recognizes $L(e)$. Since the two automata are different, recognize the same languages and have no unreachable states, there exists two words w_1, w_2 such $t'(q_0', w_1) = t'(q_0', w_2)$ but $[e_1]/{\approx} = move^*/{\approx}([q_0]/{\approx}, w_1) \neq move^*/{\approx}([q_0]/{\approx}, w_2) = [e_2]/{\approx}$ where e_1 and e_2 are any two representatives of their equivalence classes and thus $e_1 \not\approx e_2$. By definition of $\approx$, $L_p(e_1) \neq L_p(e_2)$. Without loss of generality, let $w_3 \in L_p(e_1) \setminus L_p(e_2)$. We have $w_1w_3 \in L(e)$ and $w_2w_3 \notin L(e)$ because $D_e/{\approx}$ recognizes $L(e)$, which is absurd since $t'(q_0', w_1w_3) = t'(q_0', w_2w_3)$ and D' also recognizes $L(e)$.

The previous theorem tells us that it is possible to associate to each state of an automaton for e (and in particular to the minimal automaton) a pre e' over e so that the language recognized by the automaton in the state e' is $L_p(e')$, that provides a very suggestive labelling of states.

The characterization of the minimal automaton we just gave does not seem to entail an original algorithmic construction, since does not suggest any new effective way for computing $\approx$. However, similarly to what has been done for derivatives (where we have similar problems), it is interesting to investigate admissible relations that are easier to compute and tend to produce small automata in most practical cases. In particular, in the next section, we shall investigate one important relation providing a common quotient between the automata built with pres and with Brzozowski's derivatives.

4. Read back

Intuitively, a pointed regular expression corresponds to a set of regular expressions. In this section we shall formally investigate this "read back" function; this will allow us to establish a more syntactic relation between traditional regular expressions and their pointed version, and to compare our technique for building a DFA with that based on derivatives.

In the following sections we shall frequently deal with *sets* of regular expressions (to be understood additively), that we prefer to the treatment of regular expressions up to associativity, commutativity and idempotence of the sum (ACI) that is for instance typical of the traditional theory of derivatives (this also clarifies that ACI-rewriting is only used at the top level).

It is hence useful to extend some syntactic operations, and especially concatenation, to sets of regular expressions, with the usual distributive meaning: if e is a regular expression and S is a set of regular expressions, then

$$Se = \{e'e | e' \in S\}$$

We define eS and S_1S_2 in a similar way. Moreover, every function on regular expressions is implicitly lifted to sets of regular expressions by taking its image. For example,

$$L(S) = \bigcup_{e \in S} L(e)$$

DEFINITION 37. *We associate to each item e a set of regular expressions $R(e)$ defined by the following rules:*

$$\begin{array}{rcl} R(\emptyset) & = & \emptyset \\ R(\epsilon) & = & \emptyset \\ R(a) & = & \emptyset \\ R(\bullet a) & = & \{a\} \\ R(e_1+e_2) & = & R(e_1) \cup R(e_2) \\ R(e_1e_2) & = & R(e_1)|e_2| \cup R(e_2) \\ R(e^*) & = & R(e)|e|^* \end{array}$$

R is extended to a pointed regular expression $\langle e, b\rangle$ as follows

$$R(\langle e,b\rangle) = R(e) \cup \epsilon(b)$$

Note that, for any item e, no regular expression in $R(e)$ is nullable.

EXAMPLE 38. *Since $\bullet((a+\epsilon)b^*) = \langle(\bullet a+\epsilon)(\bullet b)^*, true\rangle$ we have $R(\bullet((a+\epsilon)b^*)) = \{ab^*, bb^*, \epsilon\}$*

The parallel between the syntactic read-back function R and the semantics L_p of definition 9 is clear by inspection of the rules. Hence the following lemma can be proved by a trivial induction over e.

LEMMA 39. $L(R(e)) = L_p(e)$

COROLLARY 40. *For any regular expression e, $L(R(\bullet(e))) = L(e)$*

The previous corollary states that R and $\bullet(\cdot)$ are semantically inverse functions. Syntactically, they associate to each expression e an interesting "look-ahead" normal form, constituted (up to associativity of concatenation) by a set of expressions of the kind ae_a (plus ϵ if e is nullable), where e_a is a derivative of e w.r.t. a (although syntactically different from Brzozowski's derivatives, defined in the next section).

This look-ahead normal form (nf) has an interest in its own, and can be simply defined by structural induction over e.

DEFINITION 41.

$$\begin{array}{l} \mathit{nf}(\emptyset) = \emptyset \\ \mathit{nf}(\epsilon) = \emptyset \\ \mathit{nf}(a) = \{a\} \\ \mathit{nf}(e_1+e_2) = \mathit{nf}(e_1) \cup \mathit{nf}(e_2) \\ \mathit{nf}(e_1e_2) = \mathit{nf}(e_1)e_2 \text{ if } \nu(e_1) = \mathit{false} \\ \mathit{nf}(e_1e_2) = \mathit{nf}(e_1)e_2 \cup \mathit{nf}(e_2) \text{ if } \nu(e_1) = \mathit{true} \\ \mathit{nf}(e^*) = \mathit{nf}(e)e^* \end{array}$$

REMARK 42. *It is easy to prove that, for each e, the set $\mathit{nf}(e)$ is made, up to associativity of concatenation, only of expressions of the form a or ae_a. In particular no expression in $\mathit{nf}(e)$ is nullable!*

The previous remark motivates the following definition.

DEFINITION 43. $\mathit{nf}_\epsilon(e) = \mathit{nf}(e) \cup \epsilon(\nu(|e|))$

The main properties of nf_ϵ are expressed by the following two lemmas, whose simple proof is left to the reader.

LEMMA 44.

$$\begin{array}{l} \mathit{nf}_\epsilon(\emptyset) = \emptyset \\ \mathit{nf}_\epsilon(\epsilon) = \{\epsilon\} \\ \mathit{nf}_\epsilon(a) = \{a\} \\ \mathit{nf}_\epsilon(e_1+e_2) = \mathit{nf}_\epsilon(e_1) \cup \mathit{nf}_\epsilon(e_2) \\ \mathit{nf}_\epsilon(e_1e_2) = \mathit{nf}_\epsilon(e_1)e_2 \text{ if } \nu(e_1) = \mathit{false} \\ \mathit{nf}_\epsilon(e_1e_2) = \mathit{nf}(e_1)e_2 \cup \mathit{nf}_\epsilon(e_2) \text{ if } \nu(e_1) = \mathit{true} \\ \mathit{nf}_\epsilon(e^*) = \mathit{nf}(e)e^* \cup \epsilon(\nu(e)) \end{array}$$

THEOREM 45. $L(e) = L(\mathit{nf}_\epsilon(e))$

THEOREM 46. *For any pointed regular expression e,*

$$R(\bullet(e)) = \mathit{nf}_\epsilon(|e|) \cup R(e)$$

Proof. Let $\bullet(e) = \langle e', b'\rangle$; then $\epsilon \in R(\bullet(e))$ iff $b' = true$, iff $\nu(|e|) = true$. Hence the goal reduces to prove that $R(e') = \mathit{nf}|e| \cup R(e)$. We proceed by induction on the structure of e.

- $e = \emptyset$, $\bullet(\emptyset) = \langle\emptyset, \mathit{false}\rangle$ and $R(\emptyset) = \emptyset = \mathit{nf}(\emptyset)$
- $e = \epsilon$, $\bullet(\epsilon) = \langle\epsilon, true\rangle$ and $R(\epsilon) = \emptyset = \mathit{nf}(\epsilon)$
- $e = a$: $(\bullet(a)) = \langle\bullet a, \mathit{false}\rangle$ and $R(\bullet a) = \{a\} = \mathit{nf}(a)$
- $e = \bullet a$: $(\bullet(\bullet a)) = \langle\bullet a, \mathit{false}\rangle$ and $R(\bullet a) = \{a\} = \mathit{nf}(a) = \mathit{nf}(|\bullet a|) = \mathit{nf}(|\bullet a|) \cup R(\bullet a)$
- $e = e_1 + e_2$: let $\bullet(e_1+e_2) = \langle e_1' + e_2', b\rangle$; then

$$\begin{array}{l} R(e_1'+e_2') = \\ \quad = R(e_1') \cup R(e_2') \\ \quad = \mathit{nf}(|e_1|) \cup R(e_1) \cup \mathit{nf}(|e_2|) \cup R(e_2) \\ \quad = \mathit{nf}|e_1+e_2| \cup R(e_1+e_2) \end{array}$$

- $e = e_1e_2$. Let $\bullet(e_i) = \langle e_i', b_i'\rangle$. If $b_1' = \mathit{false}$ then $\bullet(e_1e_2) = \langle e_i'e_2, \mathit{false}\rangle$; moreover we know that e_1 is not nullable. We have then:

$$\begin{array}{l} R(e_1'e_2) = \\ \quad = R(e_1')|e_2| \cup R(e_2) \\ \quad = (\mathit{nf}(|e_1|) \cup R(e_1))|e_2| \cup R(e_2) \\ \quad = (\mathit{nf}(|e_1|)|e_2| \cup R(e_1)|e_2| \cup R(e_2) \\ \quad = \mathit{nf}(|e_1e_2|) \cup R(e_1e_2) \end{array}$$

  If $b_1' = true$ then $\bullet(e_1e_2) = \langle e_i'e_2', b_2'\rangle$; moreover we know that e_1 is nullable.

$$\begin{array}{l} R(e_1'e_2') = \\ \quad = R(e_1')|e_2| \cup R(e_2') \\ \quad = (\mathit{nf}(|e_1| \cup R(e_1))|e_2| \cup \mathit{nf}(|e_2|)) \cup R(e_2) \\ \quad = \mathit{nf}(|e_1|)|e_2| \cup \mathit{nf}(e_2) \cup R(e_1)|e_2| \cup R(e_2) \\ \quad = (\mathit{nf}(|e_1e_2|)) \cup R(e_1e_2) \end{array}$$

- $e = e_1^*$. Let $\bullet(e_1) = \langle e_i', b_i'\rangle$; then $\bullet(e_1^*) = \langle e_i'^*, true\rangle$;

$$\begin{array}{l} R(e_1'^*) = \\ \quad = R(e_1')|e_1|^* \\ \quad = (\mathit{nf}(e_1) \cup R(e_1))|e_1|^* \\ \quad = \mathit{nf}(e_1)|e_1|^* \cup R(e_1))|e_1|^* \\ \quad = \mathit{nf}(e_1^*) \cup R(e_1^*) \end{array}$$

COROLLARY 47. *For all regular expression e, $R(\bullet(e)) = \mathit{nf}_\epsilon(e)$*

To conclude this section, in analogy with what we did for the semantic function in Theorem 16, we express the behaviour of R in terms of the *lifted* algebraic constructors. This will be useful in Theorem 51.

LEMMA 48.

1. $R(e_1 \oplus e_2) = R(e_1) \cup R(e_2)$
2. $R(\langle e_1', \mathit{false}\rangle \odot e_2) = R(e_1')|e_2| \cup R(e_2)$
3. $R(\langle e_1', true\rangle \odot e_2) = R(e_1')|e_2| \cup \mathit{nf}_\epsilon(|e_2|) \cup R(e_2)$
4. $R(\langle e_1', \mathit{false}\rangle^\star) = R(e_1')|e_1^*|$
5. $R(\langle e_1', true\rangle^\star) = R(e_1')|e_1^*| \cup \mathit{nf}_\epsilon(|e_1^*|)$

Proof. Let $e_i = \langle e_i', b_i'\rangle$:

1. $$\begin{array}{l} R(e_1 \oplus e_2) = \\ \quad = R(\langle e_1', b_1'\rangle \oplus langlee_2', b_2')) = \\ \quad = R(\langle e_1' + e_2', b_1' \vee b_2'\rangle) \\ \quad = R(e_1' + e_2') \cup \epsilon(b_1' \vee b_2') \\ \quad = R(e_1') \cup R(e_2') \cup \epsilon(b_1)' \cup \epsilon(b_2') \\ \quad = R(e_1') \cup \epsilon(b_1') \cup R(e_2') \cup \epsilon(b_2') \\ \quad = R(e_1) \cup R(e_2) \end{array}$$

2. $R(\langle e_1', \mathit{false}\rangle \odot \langle e_2', b_2'\rangle) =$
$= R(\langle e_1'e_2', b_2'\rangle)$
$= R(e_1')|e_2| \cup R(e_2') \cup \epsilon(b_2')$
$= R(e_1')|e_2| \cup R(e_2)$

3. let $\bullet(e_2') = \langle e_2'', b_2''\rangle$

$$\begin{aligned} & R(\langle e_1', \mathit{true}\rangle \odot \langle e_2', b_2'\rangle) = \\ & \quad = R(\langle e_1'e_2'', b_2' \vee b_2''\rangle) \\ & \quad = R(e_1')|e_2| \cup R(e_2'') \cup \epsilon(b_2'') \cup \epsilon(b_2') \\ & \quad = R(e_1')|e_2| \cup R(\bullet(e_2')) \cup \epsilon(b_2') \\ & \quad = (R(e_1')|e_2| \cup nf(|e_2|) \cup R(e_2') \cup \epsilon(b_2') \\ & \quad = R(e_1')|e_2| \cup nf(|e_2|) \cup R(e_2) \end{aligned}$$

4. $R(\langle e_1', \mathit{false}\rangle^\star) = R(\langle e_1'^*, \mathit{false}\rangle) = R(e_1'^*) = R(e_1')|e_1^*|$

5. let $\bullet(e_1') = \langle e_1'', b_1''\rangle$; then $R(\bullet(e_1')) = R(e_1'') \cup \epsilon(b_1'') = nf_\epsilon(|e_1|) \cup R(e_1')$, and $R(e_1'') = nf(|e_1|) \cup R(e_1')$.

$$\begin{aligned} & R(\langle e_1', \mathit{true}\rangle^\star) = \\ & \quad = R(\langle e_1''^*, \mathit{true}\rangle) \\ & \quad = R(e_1'')|e_1^*| \cup \epsilon(\mathit{true}) \\ & \quad = (R(e_1') \cup dnf(|e_1|))|e_1^*| \cup \epsilon(\mathit{true}) \\ & \quad = R(e_1')|e_1^*| \cup dnf(|e_1|)|e_1^*| \cup \epsilon(\mathit{true}) \\ & \quad = R(e_1')|e_1^*| \cup nf(|e_1^*|) \end{aligned}$$

4.1 Relation with Brzozowski's Derivatives

We are now ready to formally investigate the relation between pointed expressions and Brzozowski's derivatives. As we shall see, they give rise to quite different constructions and the relation is less obvious than expected.
Let's start with recalling the formal definition.

DEFINITION 49.

$$\begin{array}{rcl} \partial_a(\emptyset) & = & \emptyset \\ \partial_a(\epsilon) & = & \emptyset \\ \partial_a(a) & = & \epsilon \\ \partial_a(b) & = & \emptyset \\ \partial_a(e_1 + e_2) & = & \partial_a(e_1) + \partial_a(e_2) \\ \partial_a(e_1e_2) & = & \partial_a(e_1)e_2 \text{ if not } \nu(e_1) \\ \partial_a(e_1e_2) & = & \partial_a(e_1)e_2 + \partial_a(e_2) \text{ if } \nu(e_1) \\ \partial_a(e^*) & = & \partial_a(e)e^* \end{array}$$

DEFINITION 50.

$$\begin{array}{rcl} \partial_\epsilon(e) & = & e \\ \partial_{aw}(e) & = & \partial_w(\partial_a(e)) \end{array}$$

In general, given a regular expression e over the alphabet Σ, the set $\{\partial_w(e) \mid w \in \Sigma^*\}$ of all its derivatives *is not* finite. In order to get a finite set we must suitably quotient derivatives according to algebraic equalities between regular expressions. The choice of different set of equations gives rise to different quotients, and hence to different automata. Since for finiteness it is enough to consider associativity, commutativity and idempotence of the sum (ACI), the traditional theory of Brzozowski's derivatives is defined according to these laws (although this is probably not the best choice from a practical point of view).

As a practical example, in Figure 3 we describe the automata obtained using derivatives relative to the expression $(ac + bc)^*$ (compare it with the automata of Figure 2). Also, note that the vertically aligned states are equivalent.
Let us remark, first of all, the heavy use of ACI. For instance

$$\partial_a((ac + bc)^*) = (\epsilon c + \emptyset c)(ac + bc)^*$$

while

$$\partial_b((ac + bc)^*) = (\emptyset c + \epsilon c)(ac + bc)^*$$

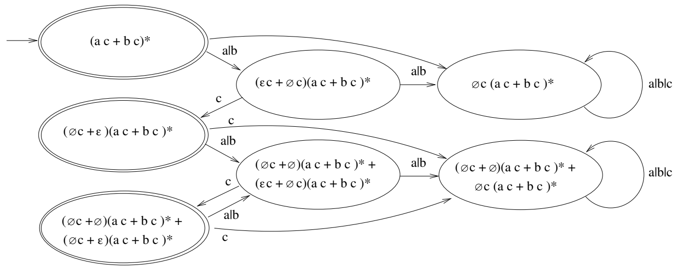

Figure 3. Automaton with Brzozowski's derivatives

and they can be assimilated only up to commutativity of the sum. As another example,

$$\begin{aligned} & \partial_a((\emptyset c + \emptyset)(ac + bc)^* + (\emptyset c + \epsilon)(ac + bc)^*) = \\ & \quad = (\emptyset c + \emptyset)(ac + bc)^* + \\ & \qquad ((\emptyset c + \emptyset)(ac + bc)^* + (\epsilon c + \emptyset c)(ac + bc)^*) \end{aligned}$$

and the latter expression can be reduce to

$$(\emptyset c + \emptyset)(ac + bc)^* + (\epsilon c + \emptyset c)(ac + bc)^*)$$

only using associativity and idempotence of the sum.

The second important remark is that, in general, it is not true that we may obtain the pre-automata by quotienting the derivative one (nor the other way round). For instance, from the initial state, the two arcs labelled a and b lead to a single state in the automata of Figure 3, but in different states in the automata of Figure 2.

A natural question is hence to understand if there exists a common *algebraic* quotient between the two constructions (not exploiting minimization).

As we shall see, this can be achieved by identifying states with a same readback in the case of pres, and states with similar look-ahead normal form in the case of derivatives.

For instance, in the case of the two automata of Figures 2 and 3, we would obtain the common quotient of Figure 4.

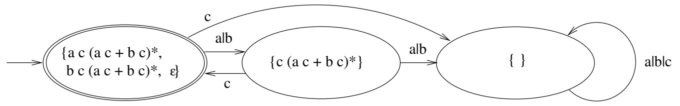

Figure 4. A quotient of the two automatons

The general picture is described by the commuting diagram of Figure 5, whose proof will be the object of the next section (in Figure 5, w obviously stands for the string $a_1 \dots a_n$).

4.2 Formal proof of the commuting diagram in Figure 5

Part of the diagram has been already proved: the leftmost triangle, used to relate the initial state of the two automata, is Corollary 47; the two triangles at the right, used to relate the final states, just states the trivial properties that $\epsilon \in R(\langle e, b\rangle)$ iff and only if $b = true$ (since no expression in $R(e)$ is nullable), and $\epsilon \in nf_\epsilon(e)$ if and only if e is nullable (see Remark 42).

We start proving the upper part. We prove it for a pointed item e and leave the obvious generalization to a pointed expression to the reader (the move operation does not depend from the presence of a trailing point, and similarly the derivative of ϵ is empty).

THEOREM 51. *For any pointed item e,*

$$R(move(e, a)) = nf_\epsilon(\partial_a(R(e)))$$

Proof. By induction on the structure of e:

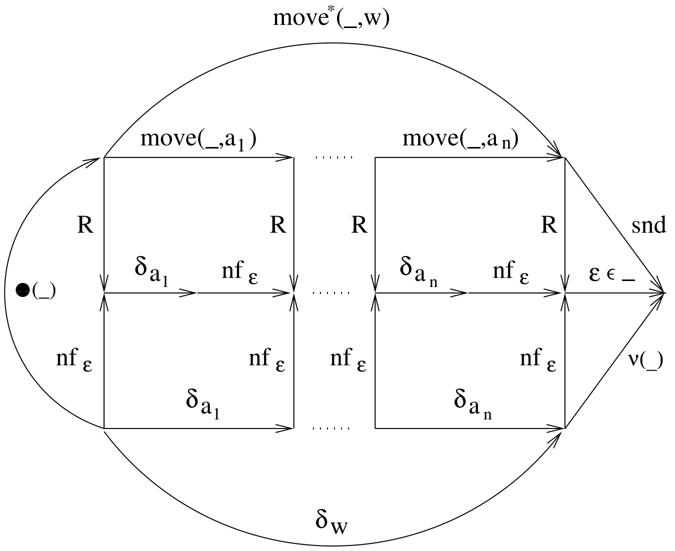

Figure 5. Pointed regular expressions and Brzozowski's derivatives

- the cases $\emptyset$, ϵ, a and b are trivial
- if $e = \bullet a$ then $\mathit{move}(\bullet a, a) = \langle a, \mathit{true}\rangle$ and $R\langle a, \mathit{true}\rangle = \{\epsilon\}$. On the other side, $\mathit{nf}_\epsilon(\partial_a(R(\bullet a)) = \mathit{nf}_\epsilon(\partial_a(\{a\})) = \mathit{nf}_\epsilon(\{\epsilon\}) = \epsilon$.
- if $e = e_1 + e_2$, then

$$\begin{array}{l} R(\mathit{move}(e_1 + e_2, a)) = \\ \quad = R(\mathit{move}(e_1, a) \oplus \mathit{move}(e_2, a)) \\ \quad = R(\mathit{move}(e_1, a)) \cup R(\mathit{move}(e_2, a)) \\ \quad = \mathit{nf}_\epsilon(\partial_a(R(e_1))) \cup \mathit{nf}_\epsilon(\partial_a(R(e_2))) \\ \quad = \mathit{nf}_\epsilon(\partial_a(R(e_1 + e_2))) \end{array}$$

- let $e = e_1 e_2$, and let us suppose that $\mathit{move}(e_1, a) = \langle e_1', \mathit{false}\rangle$ and thus $R(\mathit{move}(e_1, a) = R(e_1')$ and $\nu(\partial_a(R(e_1))) = \mathit{false}$. Then

$$\begin{array}{l} R(\mathit{move}(e_1 e_2, a)) = \\ \quad = R(\mathit{move}(e_1, a) \odot \mathit{move}(e_2, a)) \\ \quad = R(\mathit{move}(e_1, a))|\mathit{move}(e2, a)| \cup R(\mathit{move}(e_2, a)) \\ \quad = \mathit{nf}_\epsilon(\partial_a(R(e_1)))|e_2| \cup \mathit{nf}_\epsilon(\partial_a(R(e_2))) \\ \quad = \mathit{nf}_\epsilon(\partial_a(R(e_1))|e_2| \cup \partial_a(R(e_2))) \\ \quad = \mathit{nf}_\epsilon(\partial_a(R(e_1)|e_2|) \cup \partial_a(R(e_2))) \\ \quad = \mathit{nf}_\epsilon(\partial_a(R(e_1)|e_2| \cup R(e_2))) \\ \quad = \mathit{nf}_\epsilon(\partial_a(R(e_1 e_2))) \end{array}$$

If $\mathit{move}(e_1, a) = \langle e_1', \mathit{true}\rangle$ then $R(\mathit{move}(e_1, a)) = R(e_1') \cup \epsilon = \mathit{nf}_\epsilon(\partial_a(R(e_1))$. In particular $R(e_1') = \mathit{nf}(\partial_a(R(e_1))$ and $\nu(\partial_a(R(e_1))) = \mathit{true}$. We have then:

$$\begin{array}{l} R(\mathit{move}(e_1 e_2, a)) = \\ \quad = R(\mathit{move}(e_1, a) \odot \mathit{move}(e_2, a)) \\ \quad = R(e_1')|\mathit{move}(e_2, a)| \cup \mathit{nf}_\epsilon(|\mathit{move}(e_2, a)|) \cup R(\mathit{move}(e_2, a)) \\ \quad = R(e_1')|e_2| \cup \mathit{nf}_\epsilon(|e_2|) \cup R(\mathit{move}(e_2, a)) \\ \quad = \mathit{nf}(\partial_a(R(e_1)))|e_2| \cup \mathit{nf}_\epsilon(|e_2|) \cup \mathit{nf}_\epsilon(\partial_a(R(e_2))) \\ \quad = \mathit{nf}_\epsilon(\partial_a(R(e_1))|e_2|) \cup \mathit{nf}_\epsilon(\partial_a(R(e_2))) \\ \quad = \mathit{nf}_\epsilon(\partial_a(R(e_1))|e_2| \cup \partial_a(R(e_2))) \\ \quad = \mathit{nf}_\epsilon(\partial_a(R(e_1)|e_2|) \cup \partial_a(R(e_2))) \\ \quad = \mathit{nf}_\epsilon(\partial_a(R(e_1)|e_2| \cup R(e_2))) \\ \quad = \mathit{nf}_\epsilon(\partial_a(R(e_1 e_2))) \end{array}$$

- let $e = e_1^*$, and let us suppose that $\mathit{move}(e_1, a) = \langle e_1', \mathit{false}\rangle$. Thus $\epsilon \notin \mathit{nf}_\epsilon(\partial_a(R(e_1)))$. Then

$$\begin{array}{l} R(\mathit{move}(e_1^*, a)) = \\ \quad = R(\mathit{move}(e_1, a)^\star) \\ \quad = R(e_1')|e_1^*| \\ \quad = \mathit{nf}_\epsilon(\partial_a(R(e_1)))|e_1^*| \\ \quad = \mathit{nf}_\epsilon(\partial_a(R(e_1))|e_1^*|) \\ \quad = \mathit{nf}_\epsilon(\partial_a(R(e_1)|e_1^*|))) \\ \quad = \mathit{nf}_\epsilon(\partial_a(R(e_1^*))) \end{array}$$

If $\mathit{move}(e_1, a) = \langle e_1', \mathit{true}\rangle$ then $R(\mathit{move}(e_1, a)) = R(e_1') \cup \epsilon = \mathit{nf}_\epsilon(\partial_a(R(e_1))$. In particular $R(e_1') = \mathit{nf}(\partial_a(R(e_1))$ and $\nu(\partial_a(()R(e_1))) = \mathit{true}$ since $\epsilon \in \mathit{nf}_\epsilon(\partial_a(R(e_1))$. We have then:

$$\begin{array}{l} R(\mathit{move}(e_1^*, a)) = \\ \quad = R(\mathit{move}(e_1, a)^\star) \\ \quad = R(e_1')|e_1^*| \cup \mathit{nf}_\epsilon(|e_1^*|) \\ \quad = \mathit{nf}(\partial_a(R(e_1)))|e_1^*| \cup \mathit{nf}_\epsilon(|e_1^*|) \\ \quad = \mathit{nf}_\epsilon(\partial_a(R(e_1))|e_1^*|) \\ \quad = \mathit{nf}_\epsilon(\partial_a(R(e_1)|e_1^*|)) \\ \quad = \mathit{nf}_\epsilon(\partial_a(R(e_1^*))) \end{array}$$

We pass now to prove the lower part of the diagram in Figure 5, namely that for any regular expression e,

$$\mathit{nf}_\epsilon(\partial_a(e)) = \mathit{nf}_\epsilon(\partial_a(\mathit{nf}_\epsilon(e)))$$

Since however, $\mathit{nf}_\epsilon(\partial_a(\mathit{nf}_\epsilon(e))) = \mathit{nf}_\epsilon(\partial_a(\mathit{nf}(e)))$ (the derivative of ϵ is empty), this is equivalent to prove the following result.

THEOREM 52. $\mathit{nf}_\epsilon(\partial_a(e)) = \mathit{nf}_\epsilon(\partial_a(\mathit{nf}(e)))$

Proof. The proof is by induction on e. Any induction hypothesis over a regular expression e_1 can be strengthened to $\mathit{nf}_\epsilon(\partial_a(e_1)e_2) = \mathit{nf}_\epsilon(\partial_a(\mathit{nf}(e_1))e_2)$ for all e_2 since

$$\begin{array}{l} \mathit{nf}_\epsilon(\partial_a(e_1)e_2) \\ \quad = \mathit{nf}_\epsilon(\partial_a(e_1))e_2 \cup (\mathit{nf}_\epsilon(e_2) \text{ if } \nu(\partial_a(e_1)) \\ \quad = \mathit{nf}_\epsilon(\partial_a(\mathit{nf}(e_1)))e_2 \cup (\mathit{nf}_\epsilon(e_2) \text{ if } \nu(\partial_a(\mathit{nf}(e_1))) \\ \quad = \mathit{nf}_\epsilon(\partial_a(\mathit{nf}(e_1))e_2) \end{array}$$

(observe that $\nu(\partial_a(e_1)) = \nu(\partial_a(\mathit{nf}(e_1)))$ since the languages denoted by $\partial_a(e_1)$ and $\partial_a(\mathit{nf}(e_1))$ are equal).
We must consider the following cases.

- If e is ϵ, $\emptyset$ or a symbol b different from a then both sides of the equation are empty
- If e is a, $\mathit{nf}_\epsilon(\partial_a(a)) = \mathit{nf}_\epsilon(\epsilon) = \{\epsilon\} = \mathit{nf}_\epsilon(\partial_a(\{a\})) = \mathit{nf}_\epsilon(\partial_a(\mathit{nf}(a)))$
- If e is $e_1 + e_2$,

$$\begin{array}{l} \mathit{nf}_\epsilon(\partial_a(e_1 + e_2)) = \\ \quad = \mathit{nf}_\epsilon(\partial_a(e_1) + \partial_a(e_2)) \\ \quad = \mathit{nf}_\epsilon(\partial_a(e_1)) \cup \mathit{nf}_\epsilon(\partial_a(e_2)) \\ \quad = \mathit{nf}_\epsilon(\partial_a(\mathit{nf}(e_1))) \cup \mathit{nf}_\epsilon(\partial_a(\mathit{nf}(e_2))) \\ \quad = \mathit{nf}_\epsilon(\partial_a(\mathit{nf}(e_1) \cup \mathit{nf}(e_2))) \\ \quad = \mathit{nf}_\epsilon(\partial_a(\mathit{nf}(e_1 + e_2))) \end{array}$$

- If e is $e_1 e_2$ and $\nu(e_1) = \mathit{false}$,

$$\begin{array}{l} \mathit{nf}_\epsilon(\partial_a(e_1 e_2)) = \mathit{nf}_\epsilon(\partial_a(e_1)e_2) = \mathit{nf}_\epsilon(\partial_a(\mathit{nf}(e_1))e_2) = \\ \quad = \mathit{nf}_\epsilon(\partial_a(\mathit{nf}(e_1)e_2)) = \mathit{nf}_\epsilon(\partial_a(\mathit{nf}(e_1 e_2))) \end{array}$$

- If e is $e_1 e_2$ and $\nu(e_1) = \mathit{true}$,

$$\begin{array}{l} \mathit{nf}_\epsilon(\partial_a(e_1 e_2)) = \\ \quad = \mathit{nf}_\epsilon(\partial_a(e_1)e_2) \cup \mathit{nf}_\epsilon(\partial_a(e_2)) \\ \quad = \mathit{nf}_\epsilon(\partial_a(\mathit{nf}(e_1))e_2) \cup \mathit{nf}_\epsilon(\partial_a(\mathit{nf}(e_2))) \\ \quad = \mathit{nf}_\epsilon(\partial_a(\mathit{nf}(e_1)e_2 \cup \mathit{nf}(e_2))) \\ \quad = \mathit{nf}_\epsilon(\partial_a(\mathit{nf}(e_1 e_2))) \end{array}$$

– If e is e_1^*,

$$nf_\epsilon(\partial_a(e_1^*)) = nf_\epsilon(\partial_a(e_1)e_1^*) = nf_\epsilon(\partial_a(nf(e_1))e_1^*) = \\ = nf_\epsilon(\partial_a(nf(e_1)e_1^*)) = nf_\epsilon(\partial_a(nf(e_1^*)))$$

LEMMA 53. $R(e) = nf_\epsilon(R(e))$

Proof. We proceed by induction over e:

– $R(\emptyset) = \emptyset = nf_\epsilon(\emptyset) = nf_\epsilon(R(\emptyset))$
– $R(\epsilon) = \emptyset = nf_\epsilon(\emptyset) = nf_\epsilon(R(\epsilon))$
– $R(a) = \emptyset = nf_\epsilon(\emptyset) = nf_\epsilon(R(a))$
– $R(\bullet a) = \{a\} = nf_\epsilon(\{a\}) = nf_\epsilon(R(a))$
– $R(e_1 + e_2) = R(e_1) \cup R(e_2) = nf_\epsilon(R(e_1)) \cup nf_\epsilon(R(e_2)) = nf_\epsilon(R(e_1) \cup R(e_2)) = nf_\epsilon(R(e_1 + e_2))$
– $R(e_1e_2) = R(e_1)|e_2| \cup R(e_2) = nf_\epsilon(R(e_1))|e_2| \cup nf_\epsilon(R(e_2)) = nf_\epsilon(R(e_1)|e_2|) \cup nf_\epsilon(R(e_2)) = nf_\epsilon(R(e_1)|e_2| \cup R(e_2)) = nf_\epsilon(R(e_1e_2))$
– $R(e^*) = R(e)|e|^* = nf_\epsilon(R(e))|e|^* = nf_\epsilon(R(e)|e|^*) = nf_\epsilon(R(e^*))$

We are now ready to prove the commutation of the outermost diagram.

THEOREM 54. *For any pointed item e,*

$$R(move^*(e, w)) = nf_\epsilon(\partial_w(R(e)))$$

Proof. The proof is by induction on the structure of w. In the base case, $R(move^*(e, \epsilon)) = R(e) = nf_\epsilon(R(e)) = nf_\epsilon(\partial_\epsilon(R(e)))$. In the inductive step, by Theorem 52,

$$\begin{aligned} &R(move^*(e, aw)) = \\ &\quad = R(move^*(move(e, a), w) \\ &\quad = nf_\epsilon(\partial_w(R(move(e, a))) \\ &\quad = nf_\epsilon(\partial_w(nf_\epsilon(\partial_a(R(e))))) \\ &\quad = nf_\epsilon(\partial_w(\partial_a(R(e)))) \\ &\quad = nf_\epsilon(\partial_{aw}(R(e))) \end{aligned}$$

COROLLARY 55. *For any regular expression e,*

$$R(move^*(\bullet e, w)) = nf_\epsilon(\partial_w(e))$$

Proof.

$$R(move^*(\bullet e, w)) = nf_\epsilon(\partial_w(R(\bullet e)) = nf_\epsilon(\partial_w(nf_\epsilon(e)) = nf_\epsilon(\partial_w(e))$$

Another important consequence of Lemmas 51 and 52 is that R and nf_ϵ are admissible relations (respectively, over pres and over derivatives).

THEOREM 56. *$kn(R(\cdot))$ (the kernel of $R(\cdot)$) is an admissible equivalence relation over pres.*

Proof. By Lemma 39 we derive that for all pres e_1, e_2, if $R(e_1) = R(e_2)$ then $L_p(e_1) = L_p(e_2)$. We also need to prove that for all pres e_1, e_2 and all symbol a, if $R(e_1) = R(e_2)$ then $R(move(e_1, a)) = R(move(e_2, a))$. By Theorem 51

$$R(move(e_1, a)) = nf_\epsilon(\partial_a(R(e_1)) = nf_\epsilon(\partial_a(R(e_2)) = \\ = R(move(e_2, a))$$

THEOREM 57. *$kn(nf_\epsilon(e))$ is an admissible equivalence relation over regular expressions*

Proof. By Lemma 45 we derive that for all regular expressions e_1, e_2, if $nf_\epsilon(e_1) = nf_\epsilon(e_2)$ then $L(e_1) = L(e_2)$. We also need to prove that for all regular expressions e_1, e_2 and all symbol a, if $nf_\epsilon(e_1) = nf_\epsilon(e_2)$ then $nf_\epsilon(\partial_a(e_1)) = nf_\epsilon(\partial_a(e_2))$.
By Theorem 52

$$nf_\epsilon(\partial_a(e_1)) = nf_\epsilon(\partial_a(nf_\epsilon(e_1)) = nf_\epsilon(\partial_a(nf_\epsilon(e_2)) = \\ = nf_\epsilon(\partial_a(e_2))$$

THEOREM 58.
For each regular expression e, let $D_e^\bullet = (Q^\bullet, \Sigma, \bullet e, t^\bullet, F^\bullet)$ be the automaton for e built according to Definition 30 and let $D_e^\delta = (Q^\delta, \Sigma, e, t^\delta, F^\delta)$ the automaton for e obtained with derivatives. Let $kn(R)$ and $kn(nf_\epsilon)$ be the kernels of R and nf_ϵ respectively. Then $D_e^\bullet/_{kn(R)} = D_e^\delta/_{kn(nf_\epsilon)}$.

Proof. The results holds by commutation of Figure 5, that is granted by the previous results, in particular by Corollary 55, Theorem 56, Theorem 57, and the commutation of the triangles relative to the initial and final states.

Theorem 58 relates our finite automata with the infinite states ones obtained via Brzozowski's derivatives before quotienting the automata states by means of ACI to make them finite. The following easy lemma shows that $kn(nf_\epsilon)$ is an equivalence relation finer than ACI and thus Theorem 58 also holds for the standard finite Brzozowski's automata since we can quotient with ACI first.

LEMMA 59. *Let e_1 and e_2 be regular expressions. If $e_1 =_{ACI} e_2$ then $nf_\epsilon(e_1) = nf_\epsilon(e_2)$.*

5. Merging

By Theorem 16, $L_p(\bullet e) = L_p(e) \cup L(|e|)$. A more syntactic way to look at this result is to observe that $\bullet(e)$ can be obtained by "merging" together the points in e and $\bullet(|e|)$, and that the language defined by merging two pointed expressions e_1 and e_2 is just the union of the two languages $L_p(e_1)$ and $L_p(e_2)$. The merging operation, that we shall denote with a $\dagger$, does also provide the relation between deterministic and nondeterministic automata where, as in Watson [10, 11], we may label states with expressions with a single point (for lack of space, we shall not explicitly address the latter issue in this paper, that is however a simple consequence of Theorem 67). Finally, the merging operation will allow us to explain why the technique of pointed expressions cannot be (naively) generalized to intersection and complement (see Section 5.1).

DEFINITION 60. *Let e_1 and e_2 be two items on the same carrier $|e|$. The merge of e_1 and e_2 is defined by the following rules by recursion over the structure of e:*

$$\begin{array}{rcl} \emptyset \dagger \emptyset &=& \emptyset \\ \epsilon \dagger \epsilon &=& \epsilon \\ a \dagger a &=& a \\ \bullet a \dagger a &=& \bullet a \\ a \dagger \bullet a &=& \bullet a \\ \bullet a \dagger \bullet a &=& \bullet a \\ (e_1^1 + e_2^1) \dagger (e_1^2 + e_2^2) &=& (e_1^1 \dagger e_1^2) + (e_2^1 \dagger e_2^2) \\ (e_1^1 e_2^1) \dagger (e_1^2 e_2^2) &=& (e_1^1 \dagger e_1^2)(e_2^1 \dagger e_2^2) \\ e_1^* \dagger e_2^* &=& (e_1 \dagger e_2)^* \end{array}$$

The definition is extended to pres as follows:

$$\langle e_1, b_1 \rangle \dagger \langle e_2, b_2 \rangle = \langle e_1 \dagger e_2, b_1 \vee b_2 \rangle$$

THEOREM 61. *$\dagger$ is commutative, associative and idempotent*

Proof. Trivial by induction over the structure of the carrier of the arguments.

THEOREM 62. $L_p(e_1 \dagger e_2) = L_p(e_1) \cup L_p(e_2)$

Proof. Trivial by induction on the common carrier of the items of e_1 and e_2.

All the constructions we presented so far commute with the merge operation. Since merging essentially corresponds to the subset construction over automata, the following theorems constitute the proof of correctness of the subset construction.

THEOREM 63. $(e_1^1 \dagger e_1^2) \oplus (e_2^1 \dagger e_2^2) = (e_1^1 \oplus e_2^1) \dagger (e_1^2 \oplus e_2^2)$

Proof. Trivial by expansion of definitions.

THEOREM 64.

1. *for e_1 and e_2 items on the same carrier,*
$$\bullet(e_1 \dagger e_2) = \bullet(e_1) \dagger \langle e_2, \mathit{false}\rangle$$
2. *for e_1 and e_2 pres on the same carrier,*
$$\bullet(e_1 \dagger e_2) = \bullet(e_1) \dagger e_2$$
3. $(e_1^1 \dagger e_1^2) \odot (e_2^1 \dagger e_2^2) = (e_1^1 \odot e_2^1) \dagger (e_1^2 \odot e_2^2)$

COROLLARY 65.
$$\bullet(e_1 \dagger e_2) = e_1 \dagger \bullet(e_2) = \bullet(e_1) \dagger \bullet(e_2)$$

Proof.*[of the corollary] The corollary is a simple consequence of commutativity of $\dagger$ and idempotence of $\bullet(\cdot)$:*
$$\bullet(e_1 \dagger e_2) = \bullet(e_2 \dagger e_1) = \bullet(e_2) \dagger e_1 = e_1 \dagger \bullet(e_2)$$
$$\bullet(e_1 \dagger e_2) = \bullet(\bullet(e_1 \dagger e_2)) = \bullet(\bullet(e_1) \dagger e_2) = \bullet(e_1) \dagger \bullet(e_2)$$

Proof.[of 1.] We first prove $\bullet(e_1 \dagger e_2) = \bullet(e_1) \dagger \langle e_2, \mathit{false}\rangle$ by induction over the structure of the common carrier of e_1 and e_2, assuming that 3. holds on terms whose carrier is structurally smaller than e.

- If $|e_1|$ is $\emptyset$, ϵ, a, $\bullet a$ then trivial
- If e_1 is $e_1^1 + e_1^2$ and e_2 is $e_2^1 + e_2^2$:
$$\begin{aligned}&\bullet((e_1^1 + e_1^2) \dagger (e_2^1 + e_2^2)) =\\ &\quad = \bullet((e_1^1 \dagger e_2^1) + (e_1^2 \dagger e_2^2))\\ &\quad = \bullet(e_1^1 \dagger e_2^1) \oplus \bullet(e_1^2 \dagger e_2^2)\\ &\quad = (\bullet(e_1^1) \dagger \langle e_2^1, \mathit{false}\rangle) \oplus (\bullet(e_1^2) \dagger \langle e_2^2, \mathit{false}\rangle)\\ &\quad = (\bullet(e_1^1) \oplus \bullet(e_1^2)) \dagger (\langle e_2^1, \mathit{false}\rangle \oplus \langle e_2^2, \mathit{false}\rangle)\\ &\quad = \bullet(e_1^1 + e_1^2) \dagger \langle e_2^1 + e_2^2, \mathit{false}\rangle\end{aligned}$$
- If e_1 is $e_1^1 e_1^2$ and e_2 is $e_2^1 e_2^2$ then, using 3. on items whose carrier is structurally smaller than $|e_1|$,
$$\begin{aligned}&\bullet((e_1^1 e_1^2) \dagger (e_2^1 e_2^2)) =\\ &\quad = \bullet((e_1^1 \dagger e_2^1)(e_1^2 \dagger e_2^2))\\ &\quad = \bullet(e_1^1 \dagger e_2^1) \odot \langle e_1^2 \dagger e_2^2, \mathit{false}\rangle\\ &\quad = (\bullet(e_1^1) \dagger \langle e_2^1, \mathit{false}\rangle) \odot (\langle e_1^2, \mathit{false}\rangle \dagger \langle e_2^2, \mathit{false}\rangle)\\ &\quad = (\bullet(e_1^1) \odot \langle e_1^2, \mathit{false}\rangle) \dagger (\langle e_2^1, \mathit{false}\rangle \odot \langle e_2^2, \mathit{false}\rangle))\\ &\quad = \bullet(e_1^1 e_1^2) \dagger \langle e_2^1 e_2^2, \mathit{false}\rangle\end{aligned}$$
- If e_1 is e_1^{1*} and e_2 is e_2^{1*}, let $\bullet(e_1^1 \dagger e_2^1) = \langle e', b'\rangle$ and $\bullet(e_1^1) = \langle e'', b''\rangle$. By induction hypothesis, $\langle e', b'\rangle = \bullet(e_1^1 \dagger e_2^1) = \bullet(e_1^1) \dagger \langle e_2^1, \mathit{false}\rangle = \langle e'', b''\rangle \dagger \langle e_2^1, \mathit{false}\rangle$ Then
$$\begin{aligned}&\bullet(e_1^{1*} \dagger e_2^{1*}) = \bullet((e_1^1 \dagger e_2^1)^*) = \langle e'^*, \mathit{true}\rangle =\\ &\quad = \langle e''^*, \mathit{true}\rangle \dagger \langle e_2^{1*}, \mathit{false}\rangle = \bullet(e_1^{1*}) \dagger \langle e_2^{1*}, \mathit{false}\rangle\end{aligned}$$

Proof.[Of 2.] Let $\langle e_j'^i, b_j'^i\rangle = e_j^i$. By definition of $\dagger$, we have
$$e_1^1 \dagger e_1^2 = \langle e_1'^1 \dagger e_1'^2, b_1'^1 \vee b_1'^2\rangle$$
For all b and e, let $\bullet_b(e) := \begin{cases} e & \text{if } b = \mathit{false}\\ \bullet(e) & \text{otherwise}\end{cases}$
Thus for all e_1', e_2', b_1', b_2', letting $\langle e_2'', b_2''\rangle := \bullet_{b_1'}(\langle e_2', b_2'\rangle)$, the following holds:
$$\langle e_1', b_1'\rangle \odot \langle e_2', b_2'\rangle = \langle e_1' e_2'', b_2' \vee b_2''\rangle$$
Let $\langle e_2''^i, b_2''^i\rangle := \bullet_{b_1'^i}(e_2^i)$. By property 1. we have:
$$\langle e_2''^1 \dagger e_2''^2, b_2''^1 \vee b_2''^2\rangle = \bullet_{b_1'^1}(e_2^1) \dagger \bullet_{b_1'^2}(e_2^2) = \bullet_{b_1'^1 \vee b_1'^2}(e_2^1 \dagger e_2^2)$$
Thus
$$\begin{aligned}&(e_1^1 \dagger e_1^2) \odot (e_2^1 \dagger e_2^2) =\\ &\quad = \langle e_1'^1 \dagger e_1'^2, b_1'^1 \vee b_1'^2\rangle \odot (e_2^1 \dagger e_2^2)\\ &\quad = \langle (e_1'^1 \dagger e_1'^2)(e_2''^1 \dagger e_2''^2), b_2'^1 \vee b_2'^2 \vee b_2''^1 \vee b_2''^2\rangle\\ &\quad = \langle (e_1'^1 e_2''^1) \dagger (e_1'^2 e_2''^2), (b_2'^1 \vee b_2''^1) \vee (b_2'^2 \vee b_2''^2)\rangle\\ &\quad = \langle e_1'^1 e_2''^1, b_2'^1 \vee b_2''^1\rangle \dagger \langle e_1'^2 e_2''^2, b_2'^2 \vee b_2''^2\rangle\\ &\quad = (e_1^1 \odot e_2^1) \dagger (e_1^2 \odot e_2^2)\end{aligned}$$

THEOREM 66. $(e_1 \dagger e_2)^\star = e_1^\star \dagger e_2^\star$

Proof. Let $e_1 = \langle e_1^1, b_1\rangle$ and $e_2 = \langle e_2^1, b_2\rangle$. Thus
$$(\langle e_1^1, b_1\rangle \dagger \langle e_2^1, b_2\rangle)^\star = \langle e_1^1 \dagger e_2^1, b_1 \vee b_2\rangle^\star$$
Let define e', e_1' and e_2' by cases on b_1 and b_2 with the property that $e' = e_1' \dagger e_2'$:

- If $b_1 = b_2 = \mathit{false}$ then let $e_i' = e_i^1$ and $e' = e_1^1 \dagger e_2^1$. Obviously $e' = e_1' \dagger e_2'$.
- If $b_1 = \mathit{true}$ and $b_2 = \mathit{false}$ then let $\bullet(e_1^1) = \langle e_1', b_1'\rangle$, let $e_2' = e_2^1$ and let $\bullet(e_1^1 \dagger e_2^1) = \bullet(e_1^1) \dagger \langle e_2^1, \mathit{false}\rangle = \langle e', b'\rangle$. Hence $e_1' \dagger e_2^1 = e_1' \dagger e_2' = e'$.
- The case $b_1 = \mathit{false}$ and $b_2 = \mathit{true}$ is handled dually to the previous one.
- If $b_1 = \mathit{true}$ and $b_2 = \mathit{true}$ then let $\bullet(e_i^1) = \langle e_i', b_i'\rangle$ and let $\bullet(e_1^1 \dagger e_2^1) = \bullet(e_1^1) \dagger \bullet(e_2^1) = \langle e', b'\rangle$. Hence $e_1' \dagger e_2' = e'$.

In all cases,
$$\begin{aligned}&\langle e_1^1 \dagger e_2^1, b_1 \vee b_2\rangle^\star = \langle e'^*, b_1 \vee b_2\rangle = \langle (e_1' \dagger e_2')^*, b_1 \vee b_2\rangle =\\ &\quad = \langle e_1'^* \dagger e_2'^*, b_1 \vee b_2\rangle = \langle e_1'^*, b_1\rangle \dagger \langle e_2', b_2{}^*\rangle\\ &\quad = \langle e_1^1, b_1\rangle^\star \dagger \langle e_2^1, b_2\rangle^\star\end{aligned}$$

THEOREM 67. $\quad move(e_1 \dagger e_2, a) = move(e_1, a) \dagger move(e_2, a)$

Proof. The proof is by induction on the structure of e.

- the cases $\emptyset$, ϵ and $b \neq a$ are trivial by computation
- the case a has four sub-cases: if e_1 and e_2 are both a, then $move(a \dagger a, a) = \langle \emptyset, \mathit{false}\rangle = move(a, a) \dagger move(a, a)$; otherwise at least one in e_1 or e_2 is $\bullet a$ and $move(e_1 \dagger e_2, a) = move(\bullet a, a) = \langle a, \mathit{true}\rangle = move(e_1, a) \dagger move(e_2, a)$
- if e is $e^1 + e^2$ then
$$\begin{aligned}&move((e_1^1 + e_1^2) \dagger (e_2^1 + e_2^2), a) =\\ &\quad = move((e_1^1 \dagger e_2^1) + (e_1^2 \dagger e_2^2), a)\\ &\quad = move(e_1^1 \dagger e_2^1, a) \oplus move(e_1^2 \dagger e_2^2, a)\\ &\quad = (move(e_1^1, a) \dagger move(e_2^1, a)) \oplus (move(e_1^2, a) \dagger move(e_2^2, a))\\ &\quad = (move(e_1^1, a) \oplus move(e_1^2, a)) \dagger (move(e_2^1, a) \oplus move(e_2^2, a))\\ &\quad = move(e_1^1 + e_1^2, a) \dagger move(e_2^1 + e_2^2, a)\end{aligned}$$
- if e is $e^1 e^2$ then
$$\begin{aligned}&move((e_1^1 e_1^2) \dagger (e_2^1 e_2^2), a) =\\ &\quad = move((e_1^1 \dagger e_2^1)(e_1^2 \dagger e_2^2), a)\\ &\quad = move(e_1^1 \dagger e_2^1, a) \odot move(e_1^2 \dagger e_2^2, a)\\ &\quad = (move(e_1^1, a) \dagger move(e_2^1, a)) \odot (move(e_1^2, a) \dagger move(e_2^2, a))\\ &\quad = (move(e_1^1, a) \odot move(e_1^2, a)) \dagger (move(e_2^1, a) \odot move(e_2^2, a))\\ &\quad = move(e_1^1 e_1^2, a) \dagger move(e_2^1 e_2^2, a)\end{aligned}$$
- if e is e^{1*} then
$$\begin{aligned}&move(e_1^{1*} \dagger e_2^{1*}) = move((e_1^1 \dagger e_2^1)^*) =\\ &\quad = move(e_1^1 \dagger e_2^1)^\star = (move(e_1^1) \dagger move(e_2^1))^\star\\ &\quad = move(e_1^1)^\star \dagger move(e_2^1)^\star = move(e_1^{1*}) \dagger move(e_2^{1*})\end{aligned}$$

5.1 Intersection and complement

Pointed expressions cannot be generalized in a trivial way to the operations of intersection and complement. Suppose to extend the definition of the language in the obvious way, letting $L_p(e_1 \cap e_2) =$

$L_p(e_1) \cap L_p(e_2)$ and $L_p(\neg e) = \overline{L_p(e)}$. The problem is that merging is no longer additive, and Theorem 16 does not hold any more. For instance, consider the two expressions $e_1 = \bullet a \cap a$ and $e_2 = a \cap \bullet a$. Clearly $L_p(e_1) = L_p(e_2) = \emptyset$, but $L_p(e_1 \dagger e_2) = L_p(\bullet a \cap \bullet a) = \{a\}$. To better understand the problem, let $e = (\bullet ba \cap \bullet a) | \bullet b$, and let us consider the result of $move(e^*, b)$. Since $move(e, b) = \langle (b \bullet a \cap a)|b), true\rangle$, we should broadcast a new point inside $(b \bullet a \cap a)|b)$, hence $move(e^*, b) = (\bullet b \bullet a \cap \bullet a) | \bullet b)^*$, that is obviously wrong.

The problems in extending the technique to intersection and complement are not due to some easily avoidable deficiency of the approach but have a deep theoretical reason: indeed, even if these operators do not increase the expressive power of regular expressions they can have a drastic impact on succinctness, making them much harder to handle. For instance it is well known that expressions with complements can provide descriptions of certain languages which are non-elementary more compact than standard regular expression [15]. Gelade [12] has recently proved that for any natural number n there exists a regular expression with intersection of size $\mathcal{O}(n)$ such that any DFA accepting its language has a double-exponential size, i.e. it contains at least 2^{2^n} states (see also [13]). Hence, marking positions with points is not enough, just because we would not have enough states.

Since the problem is due to a loss of information during merging, we are currently investigating the possibility to exploit *colored* points. An important goal of this approach would be to provide simple, completely syntactic explanations for space bounds of different classes of languages.

6. Conclusions

We introduced in this paper the notion of pointed regular expression, investigated its main properties, and its relation with Brzozowski's derivatives. Points are used to mark the positions inside the regular expression which have been reached after reading some prefix of the input string, and where the processing of the remaining string should start. In particular, each pointed expression has a clear semantics. Since each pointed expression for e represents a state of the *deterministic* automaton associated with e, this means we may associate a semantics to each state in terms of the specification e and not of the behaviour of the automaton. This allows a *direct*, *intuitive* and *easily verifiable* construction of the deterministic automaton for e.

A major advantage of pointed expressions is from the didactical point of view. Relying on an electronic device, it is a real pleasure to see points moving inside the regular expression in response to an input symbol. Students immediately grasp the idea, and are able to manually build the automata, and to understand the meaning of its states, after a single lesson. Moreover, if you have a really short time, you can altogether skip the notion of nondeterministic automata.

Regular expression received a renewed interest in recent years, mostly due to their use in XML-languages. Pointed expressions seem to open a huge range of novel perspectives and original approaches in the field, starting from the *challenging* generalization of the approach to different operators such as counting, intersection, and interleaving (e.g. exploiting colors for points, see Section 5.1). A large amount of research has been recently devoted to the so called succinteness problem, namely the investigation of the descriptional complexity of regular languages (see e.g. [12, 13, 14]). Since, as observed in Example10, pointed expression can provide a more compact description for regular languages than traditional regular expression, it looks interesting to better investigated this issue (that seems to be related to the so called star-height [16] of the language).

It could also be worth to investigate variants of the notion of pointed expression, allowing different positioning of points inside the expressions. Merging must be better investigated, and the whole equational theory of pointed expressions, both with different and (especially) fixed carriers must be entirely developed.

As explained in the introduction, the notion of pointed expression was suggested by an attempt of formalizing the theory of regular languages by means of an interactive prover. This testify the relevance of the choice of good data structures not just for the design of algorithms but also for the formal investigation of a given field, and is a nice example of the kind of interesting feedback one may expect by the interplay with automated devices for proof development.

References

[1] G. Rozenberg and A. Salomaa, eds., *Handbook of formal languages, vol. 1: word, language, grammar*. New York, NY, USA: Springer-Verlag New York, Inc., 1997.

[2] K. Ellul, B. Krawetz, J. Shallit, and M. wei Wang, "Regular expressions: New results and open problems," *Journal of Automata, Languages and Combinatorics*, vol. 10, no. 4, pp. 407–437, 2005.

[3] J. A. Brzozowski, "Derivatives of regular expressions," *J. ACM*, vol. 11, no. 4, pp. 481–494, 1964.

[4] R. McNaughton and H. Yamada, "Regular expressions and state graphs for automata," *Ieee Transactions On Electronic Computers*, vol. 9, no. 1, pp. 39–47, 1960.

[5] S. Owens, J. H. Reppy, and A. Turon, "Regular-expression derivatives re-examined," *J. Funct. Program.*, vol. 19, no. 2, pp. 173–190, 2009.

[6] G. Berry and R. Sethi, "From regular expressions to deterministic automata," *Theor. Comput. Sci.*, vol. 48, no. 3, pp. 117–126, 1986.

[7] A. Brüggemann-Klein, "Regular expressions into finite automata," *Theor. Comput. Sci.*, vol. 120, no. 2, pp. 197–213, 1993.

[8] C.-H. Chang and R. Paige, "From regular expressions to dfa's using compressed nfa's," in *Combinatorial Pattern Matching, Third Annual Symposium, CPM 92, Tucson, Arizona, USA, April 29 - May 1, 1992, Proceedings*, vol. 644 of *Lecture Notes in Computer Science*, pp. 90–110, Springer, 1992.

[9] S. C. Kleene, "Representation of events in nerve nets and finite automata," in *Automata Studies* (C. E. Shannon and J. McCarthy, eds.), pp. 3–42, Princeton University Press, 1956.

[10] B. W. Watson, "A taxonomy of algorithms for constructing minimal acyclic deterministic finite automata," *South African Computer Journal*, vol. 27, pp. 12–17, 2001.

[11] B. W. Watson, "Directly constructing minimal dfas : combining two algorithms by brzozowski," *South African Computer Journal*, vol. 29, pp. 17–23, 2002.

[12] W. Gelade, "Succinctness of regular expressions with interleaving, intersection and counting," *Theor. Comput. Sci.*, vol. 411, no. 31-33, pp. 2987–2998, 2010.

[13] H. Gruber and M. Holzer, "Finite automata, digraph connectivity, and regular expression size," in *ICALP*, vol. 5126 of *Lecture Notes in Computer Science*, pp. 39–50, Springer, 2008.

[14] M. Holzer and M. Kutrib, "Nondeterministic finite automata - recent results on the descriptional and computational complexity," *Int. J. Found. Comput. Sci.*, vol. 20, no. 4, pp. 563–580, 2009.

[15] A. R. Meyer and L. J. Stockmeyer, "The equivalence problem for regular expressions with squaring requires exponential space," in *13th Annual Symposium on Switching and Automata Theory (FOCS)*, pp. 125–129, IEEE, 1972.

[16] L. C. Eggan, "Transition graphs and the star-height of regular events," *Michigan Mathematical Journal*, vol. 10, no. 4, pp. 385–397, 1963.